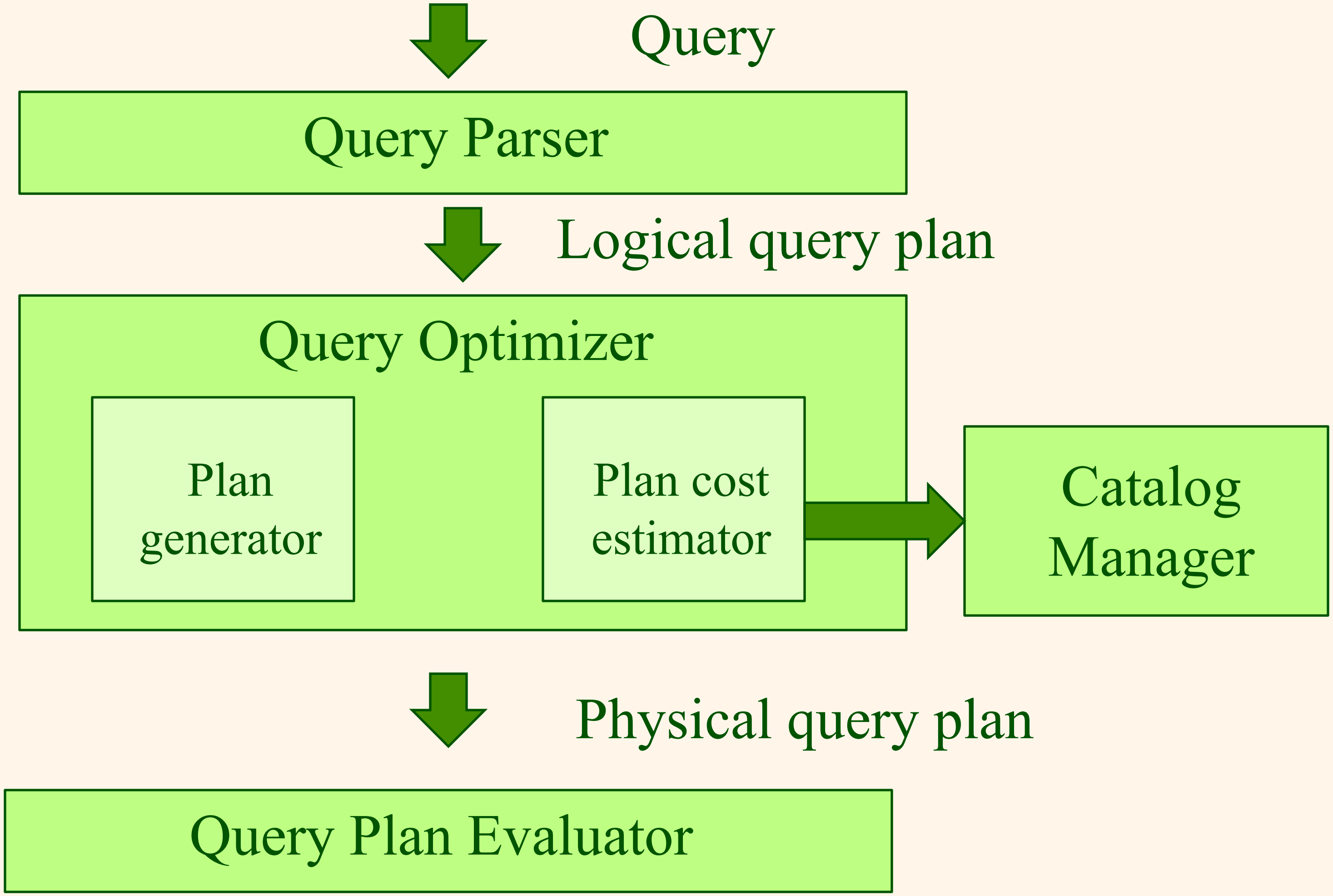
Query Optimization Wrap-up (end of Part 2 of course)

Query optimization



Parsing and decomposition

```
SELECT  S.sname,  S.age
FROM  Sailors S
WHERE  S.age =
          (SELECT  MAX(S2.age)
           FROM  Sailors S2);
```

- This query will generate two **blocks**
- Blocks correspond to a single SELECT-FROM-WHERE clause
- Blocks optimized one at a time

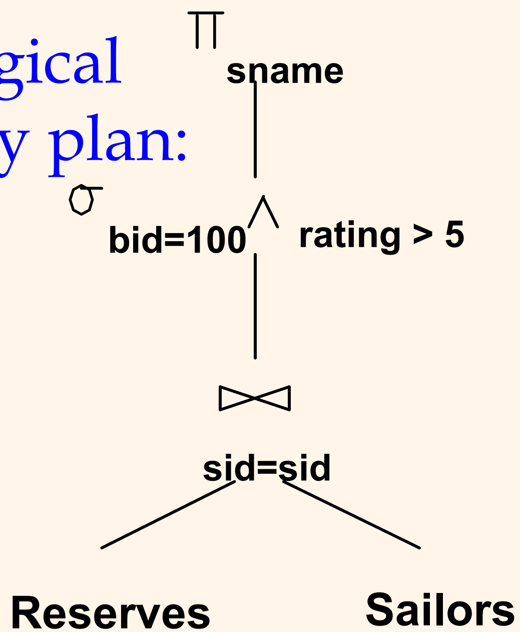
Optimizing a block

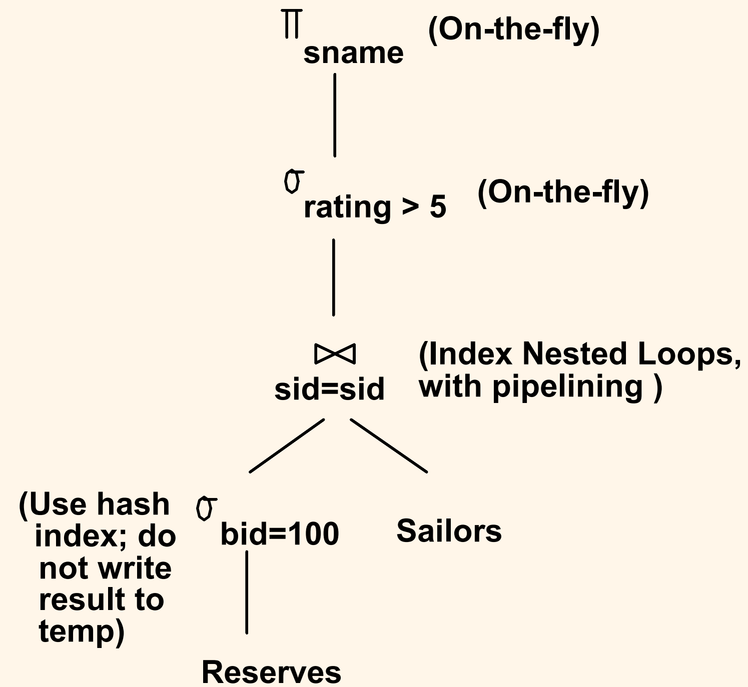
- A block is basically a Relational Algebra select-project-join ($\sigma \pi \bowtie$) expression
- With additional "operators"/annotations to handle features like
  - Aggregation
  - GROUP BY
  - ORDER BY
  - Etc
- Core of the optimizer's work: find the best **physical query plan** for the SPJ part

Query & logical and physical plans

SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
R.bid=100 AND S.rating>5

Logical query plan:





- Physical query plan = RA tree annotated with info on access methods and operator implementation

The work of an optimizer

- Generate some different physical plans
  - Reorder operators (using our handy RA equivalences)
  - Experiment with different implementations for each operator
- For each plan, estimate the cost
- Choose the lowest-cost plan

The work of an optimizer

- Ideally: Want to find best plan. Practically: Avoid worst plans!
- Optimization can't take too long, otherwise might defeat the purpose (if takes longer to optimize than to run a "dumb" plan)

Two main questions

- How to generate (a good subset of) the possible plans?
- How to compute the cost of each plan?

- Let's start with the second question….
  - Basically it's about putting together the per-operator calculations we have been doing already

Let's look at some examples

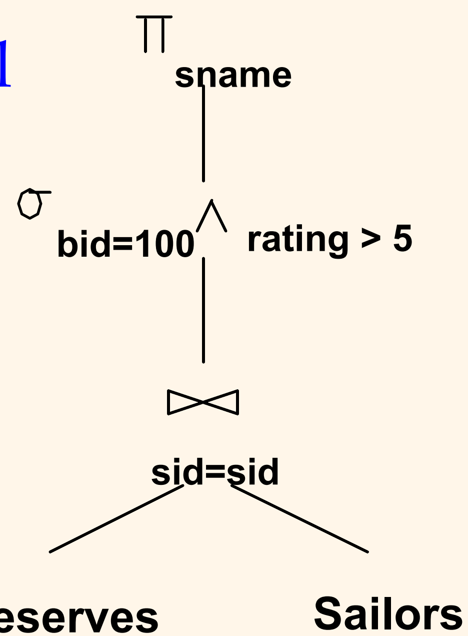
Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: date, *rname*: string)

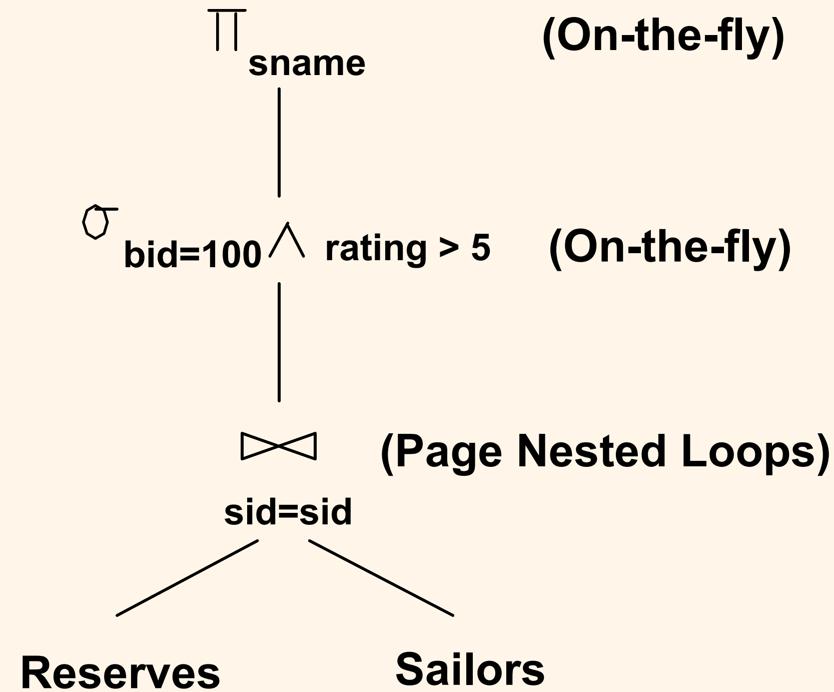
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

A first physical query plan

SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
R.bid=100 AND S.rating>5

Logical plan:



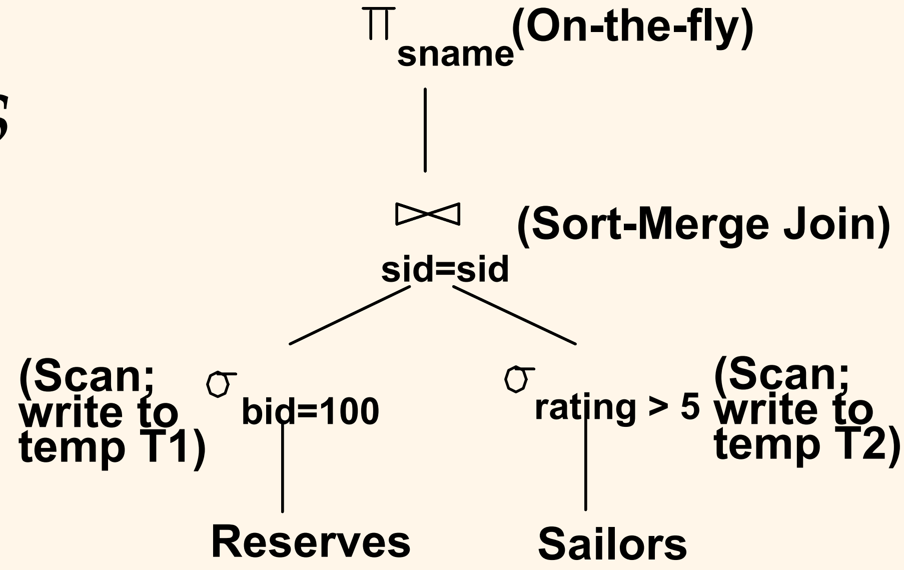


- Cost: 1000+500*1000 = 501,000 page I/Os
- Convention: **left child** of join = **outer relation**

FAQ: what does "on the fly" mean?

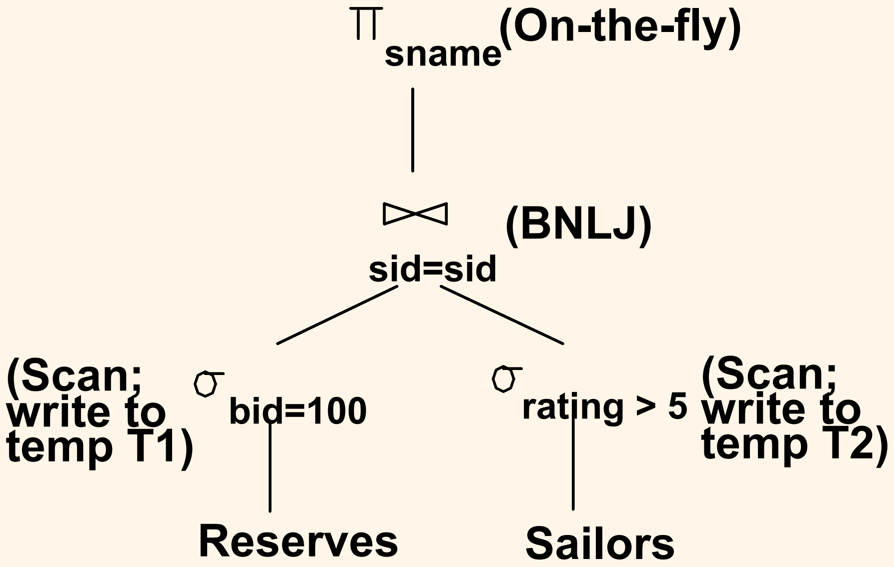
- Just means that we can apply the operation while the tuple is already in memory
- So no extra I/O cost

Alternative Plans (No Indexes)



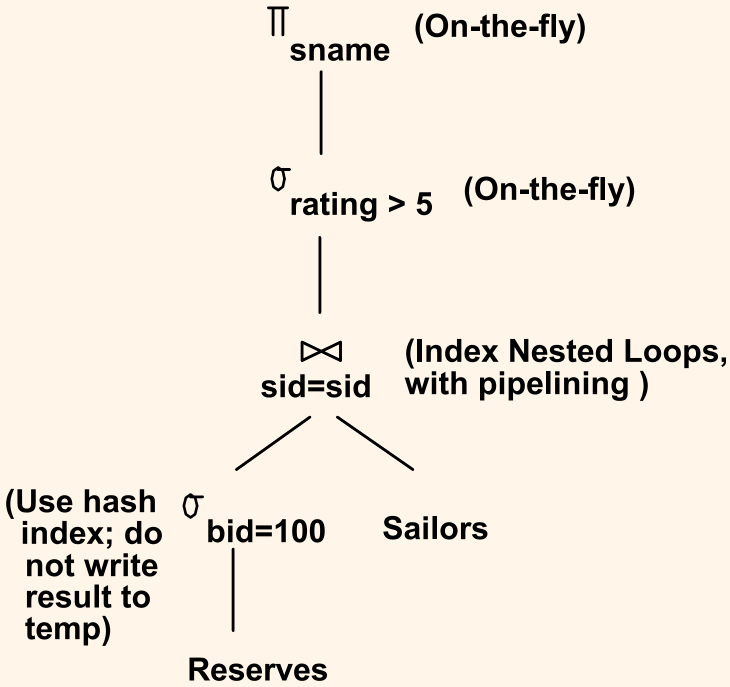
- Sort-merge join with 5 buffers:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings).
  - Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250)
  - Total: 4060 page I/Os.

Alternative Plans (No Indexes)



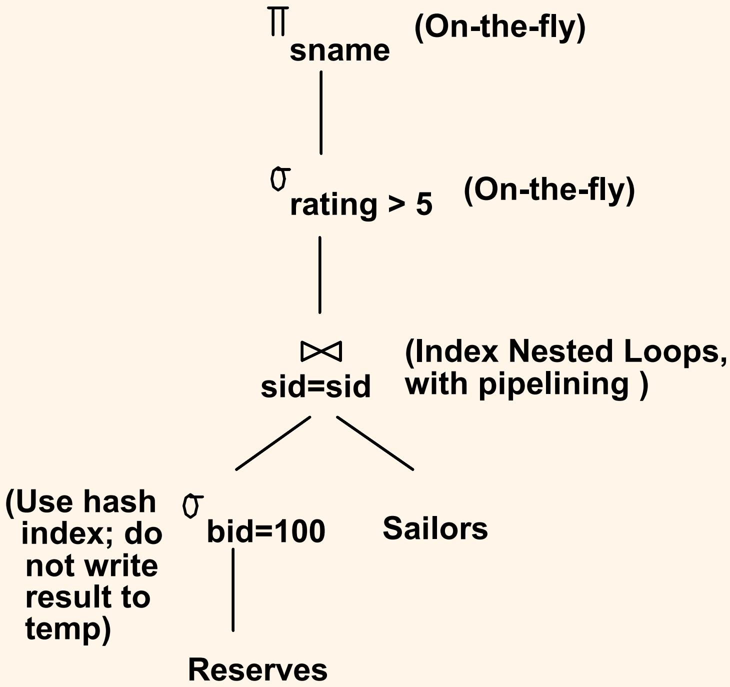
- Suppose we do BNLJ instead (3-pg blocks for T1) join cost = 10+4*250, total cost = 2770.
- If we push projections, T1 has only *sid*, T2 only *sid* and *sname*:
  - T1 fits in 3 pages, cost of BNL drops substantially, total < 2000.

Alternative Plans (With Indexes)



- Now suppose have hash index on bid of Reserves and a hash index on sid of Sailors
- INLJ with ***pipelining***
  - Outer relation in the join is never materialized
- Join column *sid* is a key for Sailors.
  - At most one matching tuple

Alternative Plans 2 With Indexes



- With clustered index on *bid* of Reserves, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages.

- Decision not to push *rating*>5 before the join is based on availability of *sid* index on Sailors.
- Cost: Selection of Reserves tuples (10 I/Os); for each, must get matching Sailors tuple (1000*1.2); total 1210 I/Os.

Calculating cost of a plan

- Cost components:
  - Reading input tables
  - Cost of each node/operator in the plan
    - Including writing intermediate tables if needed
  - Sorting result at the end if needed

Calculating cost of a plan

- Need to make assumptions about data to estimate size of intermediate tables and results
  - What fraction of the tuples will pass this selection condition?
  - How many tuples from relation R1 will join with each tuple from relation R2?

Estimating sizes

- How many tuples pass the selection in "SELECT * FROM R WHERE R.A < 10"?
- We want the reduction factor for the selection

result tuples = # tuples in R * reduction factor

- Several approaches depending on precision desired

Reduction factors

- Can pick an arbitrary reduction factor e.g. 0.1 (0.3 for inequality constraints like R.A < 42).
  - This may be enough! Remember we just want to compare plans to each other, not compute the true costs
- Can use a more precise reduction factor based on statistics/histograms

Data statistics

- DB catalogs typically contain at least:
  - # tuples and # pages for each relation.
  - # distinct key values and low/high key values for each index.
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) sometimes stored.

Reduction factors

- In our homework/exam questions we usually give you some relevant info about data distribution
  - If we do, use that info.
  - If we don't, assume some sensible reduction factor (and tell us what assumption you are making)

Reduction factors for joins

- SELECT * FROM R, S WHERE R.A = S.B
- How big is the join result?
  - Could vary from 0 to the product of |R| and |S|
- Various heuristics with various levels of precision
  - Your textbook discusses some of them (Section 15.2.1)
  - You'll explore some others in the Practicum P5
  - In an exam/homework question, watch for relevant info such as primary/foreign keys.

Enumeration of Physical Plans

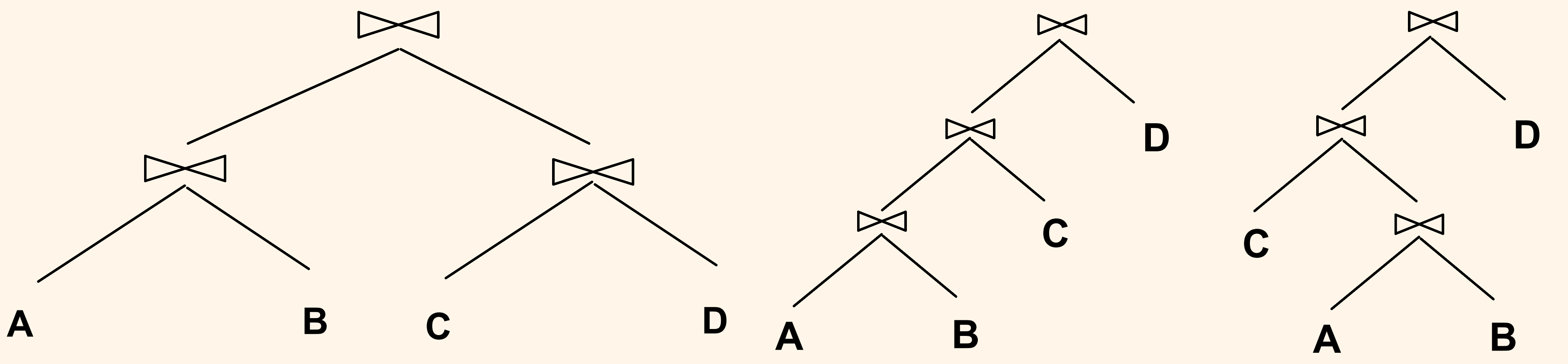
- There are two main cases:
  - Single-relation plans
  - Multiple-relation plans

Single relation queries

- No joins (by definition)
- Need to access the relation somehow
  - file scan or index
  - consider each possible access path and pick cheapest one
- Tuples are then pipelined to remaining selections, projections, aggregates.

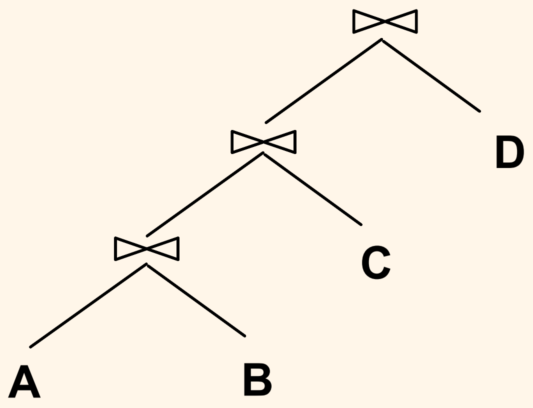
Queries Over Multiple Relations

- Core of the problem: how to evaluate the joins?
  - Can't consider all possible orderings
  - (Remember, if it takes you longer to optimize than to run the unoptimized query, that defeats the purpose …)



Queries Over Multiple Relations

- Decision: *only left-deep join trees* are considered.
  - Left-deep means the **right child of each node is a base table** (a real DB table, not a join)
    - (pushed selections/projections on base tables ok)



Queries Over Multiple Relations

- Considering only left-deep trees cuts down on search space
- Left-deep trees allow us to generate all fully pipelined plans (which are particularly desirable)
  - Outer input to join is never materialized
  - Not all left-deep plans are fully pipelined (e.g., plans that use SMJ).

Enumeration of Plans

- Left-deep plans differ only in the order of relations, the access method for each relation, and the join method for each join.

Finding the best plan

- Approach 1: exhaustive recursive search

Finding the best plan

- Consider the best plan for joining *k* relations
- There are *k* options for the last relation in the join
  - For each choice of the last relation, we want to join it with the optimal plan for the remaining *k-1* relations
- How to compute optimal plan for *k-1* relations?
  - Pick one as the last relation and recurse…

Finding the best plan

- Better approach: avoid recomputation through dynamic programming
- Save intermediate results that will be reused later
- Compute bottom-up from subsets of size 1, 2, 3, etc.

Enumeration of Plans

- Pass 1: Find best 1-relation plan for each relation
  - includes any selects/projects just on this relation.
- Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. *(All 2-relation plans.)*
- Pass k: Find best way to join result of a (k-1)-relation plan (as outer) to the kth relation. *(All k-relation plans.)*

Some practical remarks

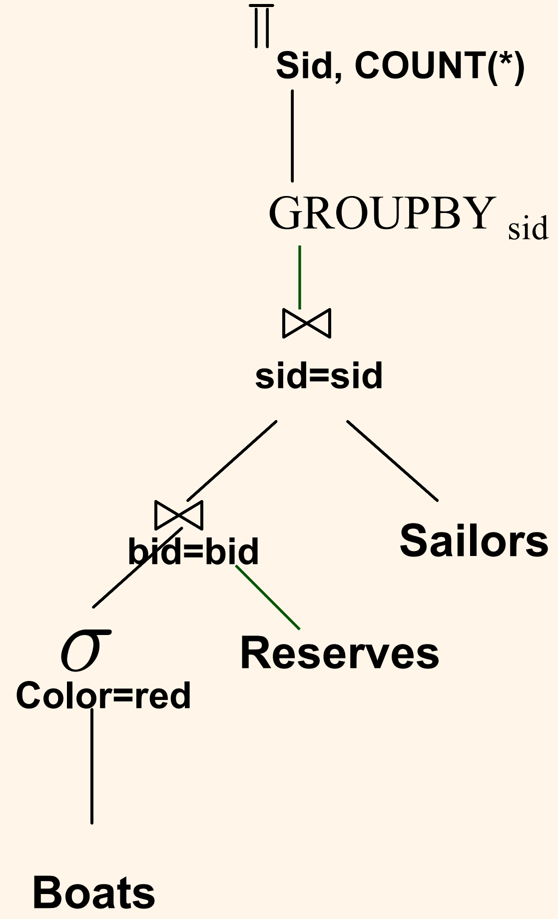
- Cost function for intermediate plans may be number of I/Os or something else
  - Simpler e.g. intermediate relation sizes
  - More complex e.g. include whether plan produces tuples in a sorted order (could be useful!!)
  - Your textbook presents an algorithm where we retain both the cheapest plan and any sorted-order plans

Enumeration of Plans (Contd.)

- ORDER BY, GROUP BY, aggregates etc. handled as a final step
  - Use a plan that gives result in sorted order
  - Or use an additional sorting operator

Example

<u>Sailors:</u>
Hash, B+ on *sid*
<u>Reserves:</u>
Clustered B+ tree on *bid*
B+ on *sid*
<u>Boats</u>
B+, Hash on *color*



SELECT S.sid, COUNT(*)
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
GROUP BY S.sid;

Pass 1

- Best plan for accessing each relation regarded as the first relation in an execution plan
  - Sailors: File Scan
    - going through B+ index gives tuples in sorted order but index is unclustered – not a good plan (file scan + sort likely better!!)
  - Reserves: File Scan
    - clustered B+ index also works but why pay the overhead of root-to-leaf navigation?
  - Boats: Hash index on color

Pass 2

- For each of the plans in pass 1, generate plans joining another relation as the inner, using all join methods
  - File Scan *Sailors* (outer) with *Reserves* (inner)
  - File Scan *Reserves* (outer) with *Sailors* (inner)
  - *Boats* hash on color (outer) with *Reserves* (inner)
  - File Scan *Reserves* (outer) with *Boats* (inner)
  - … etc
- Retain cheapest plan for each pair of relations
  - Also sorted-order plans even if they are not cheapest

A pruning heuristic

- Do not combine a partial plan with a relation unless there is a nontrivial join condition between them
  - i.e., avoid Cartesian products if possible.
  - in our case, don't bother with the Sailors/Boats pair of relations
  - note may not always be possible (e.g. if query requires cross product)

Pass 3

- For each of the plans retained from Pass 2, taken as the outer, generate plans for join with the last table
  - E.g.:
  - Outer: Boats hash on color with Reserves (bid) (sort-merge)
  - Inner: Sailors (file scan)
  - Join algorithm: sort-merge

Add cost of aggregate/GROUP BY

- Cost to sort the result by sid, if not returned sorted

Optimization summary

- Parse query into RA tree
- Optimize one block at a time
- Generate a subset of the possible evaluation plans
  - Dynamic programming approach
  - Investigates only left-deep join plans
- Cost calculations based on statistics maintained in the catalog

Optimization summary

- Parse query into RA tree
- Optimize one block at a time
- Generate a subset of the possible evaluation plans
  - Dynamic programming approach
  - Investigates only left-deep join plans
- Cost calculations based on statistics maintained in the catalog

Nested Queries

- Nested block optimized independently
- Outer block optimized with cost of "calling" nested block computation taken into account.
- Ordering of blocks means some good strategies are not considered. *The non-nested version of the query is typically optimized better.*

```
SELECT  S.sname
FROM  Sailors S
WHERE S.sid IN
   (SELECT  R.sid
    FROM  Reserves R
    WHERE R.bid=103);
```

Nested block to optimize:

```
SELECT  R.sid
FROM  Reserves R
WHERE  R.bid=103
```

Equivalent non-nested query:

```
SELECT  S.sname
FROM Sailors S, Reserves R
WHERE  S.sid=R.sid
  AND R.bid=103;
```

Nested Queries

- If the query is correlated may have to compute each nested block for every outer value
  - Algorithms for ***decorrelation***

```
SELECT  S.sname
FROM  Sailors S
WHERE EXISTS
   (SELECT  *
    FROM  Reserves R
    WHERE  R.bid=103
    AND  R.sid=S.sid);
```

Equivalent non-nested query:

```
SELECT  S.sname
FROM Sailors S, Reserves R
WHERE  S.sid=R.sid
  AND R.bid=103;
```

Demo time!

- In a real DBMS, can see the optimizer "at work"
- Let's see what indexes we have
  - SHOW INDEX FROM Sailors \G
  - \d Sailors;
- Let's add another index
- CREATE INDEX sailors_rating ON Sailors(rating) USING BTREE;
- CREATE INDEX sailors_rating ON Sailors USING BTREE (rating);

EXPLAIN statement

- EXPLAIN SELECT S.sname FROM Sailors S WHERE S.rating > 5;
- EXPLAIN SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid=R.sid;
- In PostgreSQL, handy features:
  - EXPLAIN (FORMAT JSON)
  - EXPLAIN ANALYZE

Now with more joins

- EXPLAIN SELECT S.sname FROM Sailors S, Boats B1, Boats B2, Reserves R1, Reserves R2

  WHERE S.sid=R1.sid AND R1.bid=B1.bid

  AND S.sid=R2.sid AND R2.bid=B2.bid

  AND (B1.color='red' AND B2.color='blue');

How to make your queries run faster?

- A very complex and difficult problem
- Analyze your workload and use EXPLAIN to understand what is happening
- Create indexes to help
- Understand and use the performance tuning tools your system provides
  - http://docs.oracle.com/cd/E11882_01/server.112/e41573.pdf for Oracle's Performance Tuning guide (500+ pages!!)

Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?

Understanding the Workload

- For each update in the workload:
  - Which attributes are involved in selection/join conditions?
  - How selective are these conditions likely to be?
  - What is the type of update (INSERT/DELETE/UPDATE), and what attributes are affected?

Choice of Indexes

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered? Hash/tree?

Choice of Indexes (Contd.)

- One approach: Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.

Index Selection Guidelines

- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes is important for range queries.
  - Such indexes can sometimes enable index-only strategies for important queries.
    - For index-only strategies, clustering is not important!

Index Selection Guidelines

- Try to choose indexes that benefit as many queries as possible.
- Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

Optimizer Hints

- Various DBMS will provide you with "hooks" to control how your query is evaluated

- Optimizer hints can be helpful but use them judiciously

Optimizer Hints - Caveats

- What works for one query may not work for another
- Or even for the same query when DB changes
- If you're using a lot of optimizer hints, something may be wrong
  - Are your statistics out of date?
  - Are you writing really bad SQL?
- BUT very useful for testing/experimentation purposes

MySQL index hints - example

- EXPLAIN SELECT S.sname FROM Sailors S WHERE S.rating > 5 \G

- EXPLAIN SELECT S.sname FROM Sailors S USE INDEX (sailors_rating) WHERE S.rating > 5 \G
- EXPLAIN SELECT S.sname FROM Sailors S FORCE INDEX (sailors_rating) WHERE S.rating > 5 \G

PostgreSQL

- Configuration parameters, eg. *enable_hashjoin, enable_indexonlyscan*
- Other params such as *random_page_cost* – estimate of cost of fetching random page from disk
- See documentation for more details

End of Part 2 of the course

- Query implementation and optimization
- Reached the end of material for prelim